Black or White? How to Develop an AutoTuner for Memory-based Analytics*

Mayuresh Kunjir
Duke University
mayuresh@cs.duke.edu

Shivnath Babu
Unravel Data Systems
shivnath@unraveldata.com

ABSTRACT

There is a lot of interest today in building autonomous (or, self-driving) data processing systems. An emerging school of thought is to leverage AI-driven "black box" algorithms for this purpose. In this paper, we present a contrarian view. We study the problem of autotuning the memory allocation for applications running on modern distributed data processing systems. We show that an empirically-driven "white-box" algorithm, called RelM, that we have developed provides a *close-to-optimal* tuning at a fraction of the overheads compared to state-of-the-art AI-driven "black box" algorithms, namely, Bayesian Optimization (BO) and Deep Distributed Policy Gradient (DDPG). The main reason for RelM's superior performance is that the memory management in modern memory-based data analytics systems is an interplay of algorithms at multiple levels: (i) at the resource-management level across various containers allocated by resource managers like Kubernetes and YARN, (ii) at the container level among the OS, pods, and processes such as the Java Virtual Machine (JVM), (iii) at the application level for caching, aggregation, data shuffles, and application data structures, and (iv) at the JVM level across various pools such as the Young and Old Generation. RelM understands these interactions and uses them in building an analytical solution to autotune the memory management knobs. In another contribution, called Guided-BO (GBO), we use RelM's analytical models to speed up BO. Through an evaluation based on Apache Spark, we showcase that the RelM's recommendations are significantly better than what commonly-used Spark deployments provide, and are close to the ones obtained by brute-force exploration; while GBO provides optimality guarantees for a higher, but still significantly lower cost overhead compared to the state-of-the-art AI-driven policies.

*This research is supported by NSF grant IIS-1423124.

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06...$15.00
https://doi.org/10.1145/3318464.3380591

CCS CONCEPTS

• **Information systems → Database administration**; **Autonomous database administration**.

ACM Reference Format:
Mayuresh Kunjir and Shivnath Babu. 2020. Black or White? How to Develop an AutoTuner for Memory-based Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3318464.3380591

1 INTRODUCTION

Modern data analytics systems, e.g. Spark, Tez, and Flink, are increasingly using memory both for data storage and fast computations. However, memory is a limited resource that must be managed carefully by three players:

- *Application Developer*: Judging by the magnitude of StackOverflow posts and user surveys [33, 59], 'out-of-memory' errors is a major cause of unreliable application performance. To safeguard against such errors, developers need an understanding of how much memory their application really needs and how to set the appropriate memory configurations. The prevalent rule-of-thumb to "throw more memory at your applications" is not the best approach while considering costs or the interests of other users.
- *Resource Manager*: A resource manager in a multi-tenant setting, e.g., YARN, needs to carefully allocate resources to meet the application performance goals of multiple tenants. Over-allocation leads to wasted resources and a lower throughput, while under-allocation could mean higher latency for tenants. Both problems are commonly observed in production clusters [9, 23, 51].
- *Application Platform*: The onus of ensuring a *safe* usage of memory is predominantly on the application platforms. Memory is used for various operations such as joins/aggregation, caching inputs/intermediate results, data shuffling/repartitioning, and sending intermediate/output data over network. Arbitrating memory across these operations is critical in ensuring a reliable and fast execution which is a major focus of the modern analytics platforms [32, 58].

Challenges and Contributions:

The memory management decisions in the data processing platforms are made at multiple levels (viz. the resource-management level, at the container level, at the application level, and inside the Java Virtual Machine) with complex interplays involved amongst the decisions and the performance metrics. Data analytics applications vary widely in terms of the computational model (e.g., SQL, shuffling, iterative processing) and the physical design of input data (e.g., partition sizes) translating to huge variations in their resource consumption patterns. Consequently, they exhibit complex response surfaces to configuration options relating to resource usage [12, 61]. It is seen that the default settings provided by the commonly-used system deployments leave a lot of room for improvement in terms of the reliability and the running time of the applications. Users running the applications on such deployments desire an automated tuning solution for their workload in a short span of time. Building such solutions is the focus of this paper.

The workload we consider is a data analytics application workflow along with its input data. Given the wide variety in the possible computational patterns and the physical design of data, building analytical cost-based performance models is non-trivial. Much of the previous work has focussed on training performance models *offline*, using a small-scale benchmark test bed, historical performance data, or from application performance under low workload [2, 53, 55, 60]. Offline training poses two difficulties in applying the models in real-world settings: (i) Experiments on small-scale test beds may not represent intricacies of real applications accurately; and (ii) Applying models in a changed environment or workload may involve an expensive online learning cycle. A contrasting option for tuning is an *online* search of the configuration space, typically involving a combination of random sampling and local search [5, 14, 27, 57, 61]. However, this black-box approach can be very expensive given the complex non-linear response surfaces and the high costs associated with running each experiment.

Speeding up exploration calls for an improvement-based policy which follows a Sequential Model-based Optimization (SMBO) approach [17]. SMBO iterates between fitting a surrogate model and using it to recommend the next probe. Bayesian optimization (BO) [31] is a powerful state-of-the-art SMBO technique that provides a theoretically-justified exploration of the configuration space. Another exciting possibility is to use a deep reinforcement learning approach that uses a reward-feedback approach to tuning. Deep Distributed Policy Gradient (DDPG) [28] is a powerful technique providing a model-free, actor-critic algorithm which can operate on continuous action (configuration) spaces.

We approach the tuning problem by developing a deep understanding of the internal memory management options.

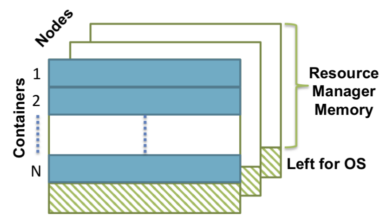

Figure 1: Memory managed by Resource Manager

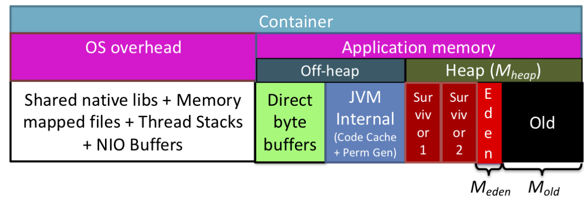

Figure 2: Container memory managed by JVM

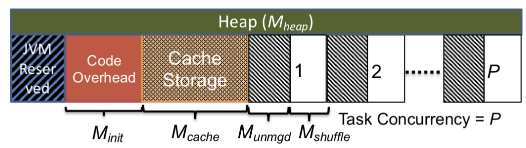

Figure 3: Heap managed by application framework

Rather than directly modeling the high level tuning objectives, such as latency, we model the impact of the memory configurations on the efficiency of the system resource utilization and the reliability of execution. This understanding is used to develop an algorithm, called RelM, that quickly tunes the memory management options using a very small number (one or two) of profiled application runs. At the core of RelM is a set of simple analytical models that estimates the requirements of the various competing memory pools within an application. Using the models, RelM guarantees a *safe*, that is, free of out-of-memory errors and, simultaneously, highly resource-efficient configuration.

In another contribution, we use RelM's analytical models to speed up the black-box tuning of BO. This modification, called *Guided Bayesian Optimization* (GBO), plugs in metrics derived from an application profile relating to reliability, efficiency, and performance overheads to the BO model.

The two solutions we have designed for tuning memory management decisions in data analytics both improve the state-of-the-art significantly and also present an interesting trade-off to the end user: While RelM offers a *good* (performing within top 5 percentile of the exhaustively searched configurations) tuning recommendation with a minimal training overhead, GBO guarantees optimality given an allowance for a slightly higher overhead. The reinforcement learning approach (DDPG) is shown to possess a great ability to adapt to high dimensional spaces as well as to changes in the test environment thereby making a strong case for use in other related auto-tuning problems.

2 PROBLEM OVERVIEW

2.1 Memory-based Analytics

Data analytics clusters employ a resource manager, such as Yarn [52], to allocate cluster resources to applications. Each application is provided with a set of *containers* by the resource manager. A container is simply a slice of physical resources carved out of a node allocated exclusively to the application. Figure 1 shows how cluster memory is allocated to multiple containers. Many popular data analytics systems (e.g., Spark, Flink, and Tez) use a JVM-based architecture. These systems launch a JVM process inside each allocated container. As shown in Figure 2, the container memory is divided into two parts: (a) Memory available to the JVM process, and (b) An overhead space for OS process management. JVM further divides its allocation into a heap space and an off-heap space. All objects, except native byte buffers, created by the application code are allocated on heap and are managed by the JVM's generational heap management [37].

Applications written in JVM languages do not explicitly allocate and free memory. Instead, the JVM periodically runs a process of garbage collection (GC in short) that frees up unreferenced objects from Heap. We focus on the default GC policy, called ParallelGC. ParallelGC uses two memory pools: Young generation and Old generation. As the names suggest, the Young pool stores newly created objects and the Old pool stores long-living objects. Young pool is split into one *Eden* space and two *Survivor* spaces only one of which is occupied at any given time. Newly created objects go to Eden first. When Eden is filled up, a collection called *Young GC* is triggered to collect unreferenced objects from Eden and the occupied Survivor. Objects that have *aged enough* [1] are moved to the Old pool. When a Young GC process finds an almost full old generation, it triggers a *Full GC* process which collects all unreferenced objects from Old and compacts the Old pool. Key tuning options controlling the time spent by GC processes are related to the sizes of the pools. Parameter *NewRatio* sets the ratio of the capacity of Old to the capacity of Young. The capacity of Eden within Young is decided by parameter *SurvivorRatio* which gives the ratio of the capacity of Eden to the capacity of a Survivor space.

Figure 3 shows Heap from the application's perspective. Except for the space reserved for the JVM's internal objects and a survivor space, the Heap can be broadly categorized into three pools:

1. *Code Overhead*: Memory required for application code objects (M_i). Treated as a constant overhead.
2. *Cache Storage*: Memory used to store the data cached by application (M_c). In particular, storing intermediate results in memory is beneficial during iterative computations.
3. *Task Memory*: The number of tasks running concurrently is set by a configuration parameter: Task Concurrency. Each task needs memory for: (a) Shuffle processing tasks such as sort and aggregation (M_s), (b) Input data objects and serialization/deserialization buffers (M_u).

[1] Aging is determined by GC parameters 'InitialTenuringThreshold' and 'MaxTenuringThreshold' [37]

Table 1: Parameters controlling memory pools across multiple levels: Container, Application Framework, and JVM displayed in order from top to bottom.

Parameter	Description	Pool(s) controlled
Heap Size	Heap size in a container	Heap (M_h)
Cache Capacity	Cache storage as a fraction of Heap	Cache Storage (M_c)
Shuffle Capacity	Shuffle memory as a fraction of Heap	Task Shuffle (M_s)
Task Concurrency	Number of tasks running concurrently	Task Unmanaged (M_u)
NewRatio	Ratio of Old capacity to Young capacity	Old (M_o)
SurvivorRatio	Ratio of Eden capacity to Survivor space	Eden (M_e)

Allocation to the pools Cache Storage (M_c) and Task Shuffle (M_s) is controlled by application frameworks both to make an efficient and error-free use of available memory. Spark, for example, provides a configuration option called *spark.memory.fraction* to bound the two pools [40]. The other two memory pools Code Overhead (M_i) and Task Unmanaged (M_u), however, are not managed explicitly. In summary, Table 1 lists the parameters controlling usage of memory pools in—and effectively impacting the performance of—memory-based analytics systems.

2.2 Application Tuning

Performance of the memory-based analytics workloads is largely dependent on the *safety* and *efficiency* of memory usage. Under this framework, an application can be tuned at the following levels: (a) while allocating resources from the resource manager, (b) while setting options provided by the application framework related to the internal memory pools, and (c) while configuring JVM parameters related to garbage collection. Applications we consider for tuning constitute a given workflow (or query plan) and a given input data. Re-using tuning results when any of these inputs changes is left out of the scope of this paper. We first outline three broad categories of tuning approaches possible for our problem setup before describing our solution.

I. Robust defaults: Cloud vendors and application frameworks provide default settings for parameters that are expected to generalize towards a broad spectrum of applications. Amazon's popular cloud-based offering Elastic MapReduce (EMR) provides a default policy for resource allocation on Spark clusters, called *MaxResourceAllocation* [34]. This policy creates a single resource container on each worker node allocating it the entire compute and memory resources. Frameworks such as Spark and Flink provide default settings for application level and JVM level memory pools [35, 40].

They use heuristics that generalize well, e.g., Old pool size is set higher than Cache Storage in order to fit the long living cache objects in the tenured space. However, the defaults leave a lot of scope for performance improvements which can be exploited easily by expert users [21, 47].

II. White-box modeling: One approach towards building an automated tuning solution is to develop analytical *What-If* models for performance estimations [15, 24, 51]. But developing such models is nontrivial [56] or downright impossible given the wide variety in the computational models and the physical design of data to consider. Most of the literature has focussed on training ML-based performance models using either a small-scale benchmark test bed, historical performance data, or from application performance under low workload [2, 13, 29, 53–55, 60]. However, the understanding developed by these *offline* approaches may not directly help tune a new application, or may potentially involve a long *online* learning cycle.

III. Black-box modeling: Search-based black-box tuning approaches to find the optimal configuration [5, 14, 27, 57, 61] are often expensive given the complex non-linear response surfaces and the high costs associated with running each experiment. A better option is to employ an improvement-based policy which follows a Sequential Model-based Optimization (SMBO) [17]. SMBO iterates between fitting a surrogate model and using it to recommend the next probe of the configuration space. Bayesian optimization (BO) [31] is a powerful state-of-the-art SMBO technique that is applied to varied designs including Database systems [2, 12], Streaming [20], Storage systems [6], and Cloud infrastructures [3, 16]. We consider BO as a candidate black-box policy for our problem. Another popular AI-based policy we consider is Deep Deterministic Policy Gradient (DDPG) [28]. It provides a powerful reinforcement learning algorithm that is hugely popular in the fields of robotics and imaging and has recently been adopted in database systems [26, 60].

Our evaluation shows that despite the advances in the black-box tuning approaches, the number of experiments (test runs) required to have sufficient confidence in predictions could still be significant. This number could be lowered if we could use some internal understanding of the impact of the memory configurations. We carry out an empirical study in Section 3 to develop a deep understanding of the various interactions among the configuration options and the resource usage metrics. This study is used in building an analytical algorithm, called RelM, to recommend a configuration that is both reliable as well as resource-efficient. RelM relies on a single application profile to learn application-specific requirements of the resource requirements for different processing needs. The requirements are fed to a set of analytical models which combine, in quick time, to recommend a setup most suited to the application's needs.

In another important contribution, the analytical models developed in RelM are used to speed-up BO. The idea is to plug-in the system internal knowledge in the form of a small number of analytical models as extra parameters to the surrogate model of BO. These parameters, in turn, help the model learn the distinction between the undesired (expensive) regions and the desired regions of the configuration space in quick time.

3 UNDERSTANDING INTERACTIONS

Data analytics applications vary widely in terms of their computational model (e.g., SQL, shuffling, iterative processing) and physical design of input data (e.g., partition sizes). This translates to variations in resource consumption patterns of the computations. We have listed the most important memory configuration options in Table 1. Here, we explore the impact of each option using the five representative benchmark applications listed in Table 2. The test suite covers a broad spectrum of computational models and physical designs making it ideal for the empirical study. All experiments were carried out on Cluster A listed in Table 3.

3.1 Containers per Node

As shown in Figure 1, physical memory on a node is divided into multiple containers by the resource manager. This creates a spectrum of choices from using a small number of *fat* containers to a large number of *thin* containers. Amazon EMR's *MaxResourceAllocation* policy creates one fat container on each node assigning it the entire node memory (minus OS overheads). We vary the number of containers on a node from 1 to 4. The corresponding Heap allocation shrinks from 4404MB to 1101MB proportionately. The other parameters are set to their default values as listed in Table 4.

Figure 4 shows the results. From the runtimes (normalized to the runtimes on the default setup), it can be noticed that WordCount and SortByKey perform significantly better on thin containers. Both the applications do not use any cache storage and are, therefore, less memory-bound compared to the ML applications, namely, K-means and SVM. However, the performance does not scale linearly because of the CPU and Disk bottlenecks. Tasks running K-means and SVM are given less memory for processing because of cache storage. As a result, thin containers run into memory pressures leading to a degradation of performance. K-means, in fact, runs into *out-of-memory* failures with 4 containers per node.

Observation 1: *Containers should be adequately sized to just meet the cache and the task memory requirements.*

Failure cases. Results presented in Figure 4 do not include PageRank application because it fails under each setup. In our technical report [22], we probe more instances of such failures. Broadly, two types of failures are observed: (a) *Out-of-memory* errors while creating objects on heap for either input data deserialization or network buffers; (b) Resource

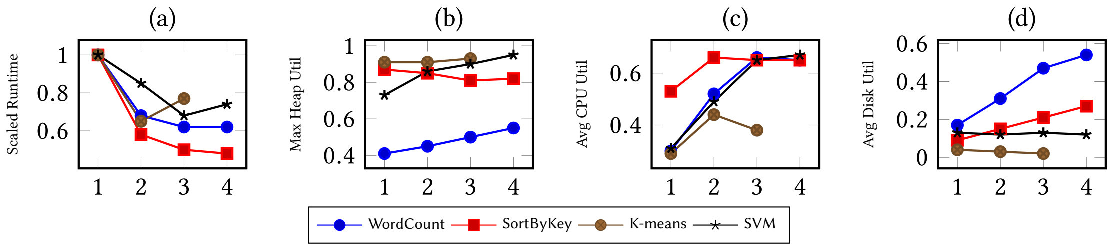

Figure 4: Impact of increasing number of containers per node on runtime (a), maximum heap utilization (b), average CPU utilization (c), and average disk utilization (d). Missing points correspond to instances of failures.

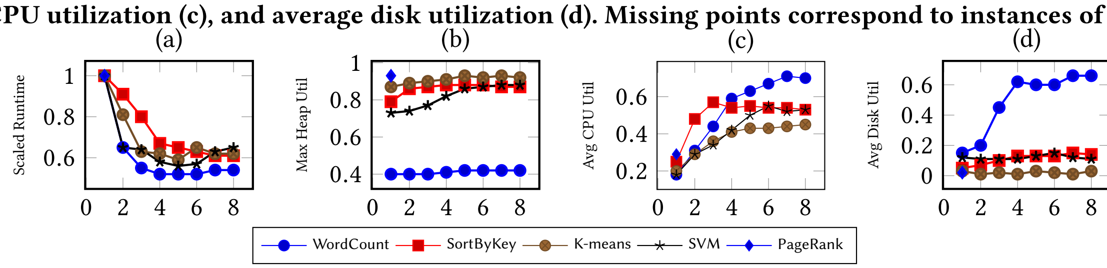

Figure 5: Impact of Task Concurrency on runtime (a), maximum heap utilization (b), average CPU utilization (c), and average disk utilization (d). PageRank **runs out of memory for Task Concurrency** ≥ 2.

Table 2: Test suite used in evaluation

Application	Category	Dataset	Partition Size
WordCount	Map and Reduce	Hadoop RandomTextWriter (50GB)	128MB
SortByKey	Map and Reduce	Hadoop RandomTextWriter (30GB)	512MB
K-means	Machine Learning	HiBench huge (100M samples)	128MB
SVM	Machine Learning	HiBench huge (100M examples)	32MB
PageRank	Graph	LiveJournal [25] (69M edges)	128MB
TPC-H	SQL	TPC-H DBGen (50 scale factor)	128MB

Table 3: Evaluation cluster setups

	Cluster A	Cluster B
Node types	Physical	Virtual EC2
Number of nodes	8	4
Memory per node	6GB	32GB
CPU cores per node	8	31 ECU
Network bandwidth	1Gbps	10Gbps
Compute Framework	Spark-2.0.1	
Resource Manager	Yarn-2.7.2	
JVM Framework	OpenJDK-1.8.0	

Table 4: Config values suggested by *MaxResourceAllocation* and framework defaults on Cluster A.

Containers per Node	1
Heap Size	4404MB
Task Concurrency	2
Cache Capacity + Shuffle Capacity	.6
NewRatio	2
SurvivorRatio	8

manager *killing* containers that exceed a preset limit for physical memory usage. Both cases are identified to be caused by a small amount of memory left for the unmanaged task objects after provisioning for the other pools.

Observation 2: *Over-provisioning for internal memory pools can result in unreliable performance.*

3.2 Task Concurrency

An important optimization to increase throughput is to increase task concurrency. We analyze this in Figure 5. The runtimes are normalized to the setup with task concurrency set to 1. The performance of each application is seen to improve with concurrency before it plateaus. For all applications except WordCount, the effect can be explained by memory pressures indicated by the max heap utilization. As each concurrently running task has to compete for a fixed sized heap, increasing task concurrency leads to more GC overheads, curtailing the benefits of the increased parallelism. Tasks for WordCount, though not bottlenecked by memory, suffer from CPU and disk bottlenecks.

Observation 3: *Resource bottlenecks including CPU, I/O, and memory must be considered while setting Task Concurrency.*

3.3 Cache and Shuffle memory

We explore impact of the memory allocated to the internal memory pools of Cache Storage and Task Shuffle in Figure 6. Since Spark uses a unified memory pool [41] to manage both, we vary a single parameter that changes the fraction of heap allocated to the unified pool. Further, we single out the applications K-means, SVM, and PageRank for the analysis of Cache Capacity as they predominantly use cache. Applications WordCount and SortByKey on the other hand, are analyzed for the shuffle memory since they use the unified memory pool exclusively for shuffle objects.

It can be noticed that an increase in Cache Capacity results in performance gains for each of the K-means, SVM, and PageRank before either the performance plateaus or containers run out of memory. We include a plot showing *Cache Hit Ratio* which gives a ratio of the number of data partitions found in cache over the total number of partitions requested to be cached. It shows SVM can fit 100% partitions in cache with a capacity over 0.5, the point where its performance plateaus. K-means hits the memory bottleneck before it can attain a ratio of 1. GC overheads, derived by averaging the fraction of time spent by tasks in GC processes, also indicate a sharp rise before containers fail at a Cache Capacity of 0.8.

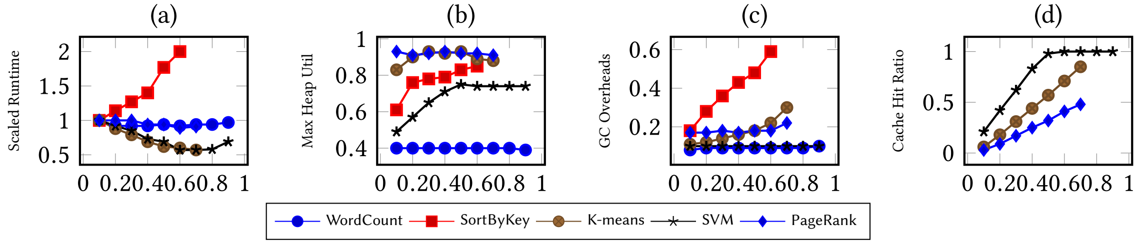

Figure 6: Impact of Cache Capacity and Shuffle Capacity on runtime (a), maximum heap utilization (b), and average per task GC Overheads (c). X-axis represents Shuffle Capacity as a fraction of Heap **for** WordCount **and** SortByKey**. On other applications, it represents Cache Capacity. The cache hit ratio is displayed in plot (d).**

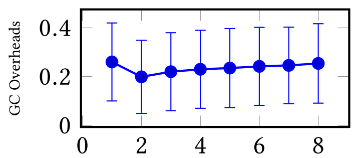

Figure 7: Impact of NewRatio on GC Overheads for K-means **with a Cache Capacity of 0.6. Error bars indicate standard deviation.**

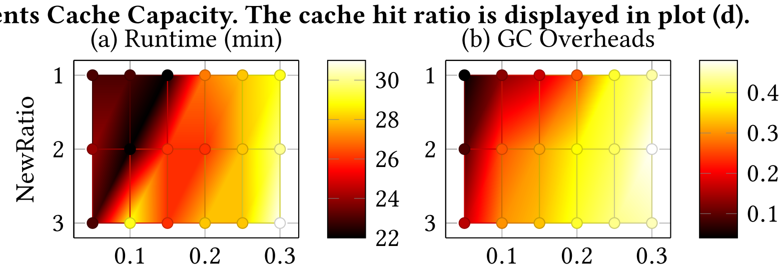

Figure 8: Impact of NewRatio and Shuffle Capacity on runtime (a) and GC Overheads (b) for SortByKey

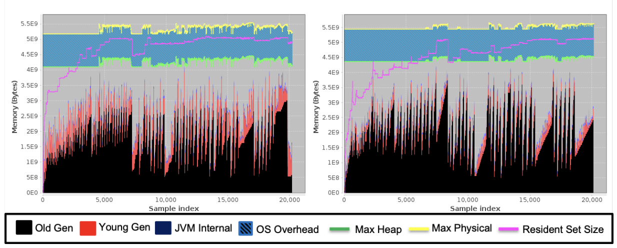

Figure 9: Comparing memory usage timeline for a container having NewRatio=2 (left) with a container having NewRatio=5. The left side configuration is more prone to failures due to physical memory usage exceeding limit set by resource manager.

Observation 4: *Leave sufficient memory for tasks while optimizing for cache storage.*

Analysis of shuffle memory throws the most counter-intuitive result for SortByKey where assigning more shuffle memory leads to performance degradation. Tasks running the reduce stage of SortByKey use memory for the in-memory sort operation. If the allocation is insufficient, tasks use an external merge-sort by spilling partially sorted records to disk and merging them later. Although increasing shuffle memory leads to lowering the number of spills, increased size of each spill puts more pressure on GC. Tasks can be seen to spend 60% time doing GC when the Shuffle Capacity is 0.6. We reason this further in the next subsection.

3.4 Interactions with GC settings

JVM's heap organization corresponds well to the cache requirements: As the cached objects reside in memory for a long time, they are served the best off the Old pool of JVM [40]. In [22], we study the interaction between Old size and Cache Capacity. The key result is presented next.

Observation 5: *Sizing* Old *smaller than* Cache Storage *can lead to huge GC overheads.*

The analysis above tells us to set Old size higher than Cache Storage but how high should it be? It turns out, high values lead to increased GC overheads due to the frequent collections. Figure 7 analyzes K-means with a Cache Capacity of 0.6 with NewRatio increased from 1 to 8. Setting NewRatio to 2 provides the best outcome since it *just* fits the cache.

The higher NewRatio settings, despite adding GC overheads, can help prevent containers exceeding the physical memory usage limit set by the resource manager. Referring to Figure 9, low value of NewRatio implies a lower frequency of GC which results in on-heap references to the objects created in off-heap space (e.g., Native ByteBuffers used in network transfers) getting collected less frequently. It causes the physical memory usage (*magenta* line) to grow more rapidly, and in some cases, exceeding the maximum physical memory cap (*yellow* line).

Observation 6: Old *capacity values larger than* Cache Storage *present a trade-off between performance and reliability.*

The shuffle memory use case is very different to the cache storage. While the cached objects have a long life, shuffle objects have a very short time span since tasks repeatedly spill the partially aggregated/ sorted results to disk. Setting Shuffle Capacity larger than Eden pool size necessitates a *full GC* every time a task spills. Figure 8 plots the runtimes and the GC overheads for SortByKey executed with Shuffle Capacity ranging from 0.05 to 0.3. The NewRatio value is increased from 1 to 3 causing the Eden capacity to go down from 37% to 18% of Heap. It should be noted that Eden contains not only the shuffle objects but also other task objects,

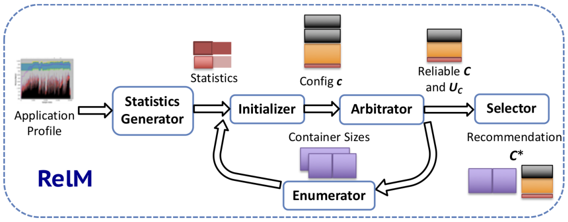

Figure 10: Application tuning process in RelM

making it hard to estimate its occupancy. A good heuristic could be to set the shuffle memory to 50% of Eden.

Observation 7: *Shuffle Capacity larger than (50% of)* Eden *can lead to huge GC overheads.*

4 RELM TUNER

The goal of RelM tuner is to recommend a setup of memory pools which ensures a reliable and fast performance. In particular, RelM meets the following objectives:

(1) Safety: Resource usage should be within allocation.

(2a) High task concurrency: Maximize the number of concurrently running tasks after ensuring (1).

(2b) High cache hit ratio: Provision sufficient memory for cache storage after ensuring (1).

(3) Low GC overheads: Limit the time spent by tasks in GC processes after ensuring (1), (2a), and (2b).

The criteria suggest a priority of goals. Safety is of foremost concern as it has the highest implications to the application performance. We rank the goals (2a) and (2b) at the same level. Depending on application characteristics, performance is primarily a function of either one of them or both (Section 3). While the former is constrained by each of the CPU, memory, I/O bottlenecks, the latter is constrained by the memory provisioned alone. The goal of *low GC overheads* is ranked the lowest in the scheme of things: Based on the settings used to meet high priority goals, we tune the parameters affecting GC overheads.

It should be noted that we do not pursue a goal of lowering shuffle data spillage here. Based on an extensive empirical study carried out by Iorgulescu et. al. [18] on Hadoop, Spark, Flink, and Tez frameworks—in addition to our own evaluation presented in Section 3—it is evident that the memory provisioned for data shuffle has a limited positive impact on application runtime. Moreover, high values for shuffle memory could lead to GC bottlenecks as shown in Section 3.4. We avoid these overheads by tuning shuffle memory and GC pool settings together as part of the goal (3).

RelM relies on a profile of application run to understand the resource requirements. The statistics derived from this run are used in evaluation of all combinations of container sizes, application memory pools settings, and JVM configurations using analytical modeling. We first comment on the container sizes we enumerate. We support multiple homogeneous containers carved out of a single node with the node memory distributed equally as shown in Figure 1. This gives us a small finite number of container size configurations.

Table 5: Statistics derived from an application profile

Notation	Description	Example
N	Containers per Node	1
M_h	Heap size	4404MB
CPU_{avg}	Average CPU usage	35%
$Disk_{avg}$	Average disk usage	2%
M_i	Code Overhead 90%ile value	115MB
M_c	Cache Storage 90%ile value	2300MB
M_s	Task Shuffle 90%ile value	0MB
M_u	Task Unmanaged 90%ile value	770MB
P	Task Concurrency	2
H	Cache Hit Ratio (the fraction of cached data partitions actually read from cache)	0.3
S	Data Spillage Fraction (the fraction of shuffle data spilled to disk)	0

Example. *Amazon EMR's m4.large nodes set the maximum memory for resource manager to 6GB with a minimum allocation size of 1GB. The possible container configurations in this case, listed as (Containers per Node, Heap Size), are: (1, 4404MB), (2, 2202MB), (3, 1468MB), and (4, 1101MB). Rest of the memory is left for OS overheads.*

Figure 10 describes the steps in tuning.

1. Application profile is processed by the *Statistics Generator* to derive a set of statistics listed in Table 5. (Section 4.1)
2. The *Enumerator* module runs each container size configuration through *Initializer* and *Arbitrator.*
3. Given a container size and the statistics from application profile, the *Initializer* module sets initial settings for memory pools optimizing each pool independently. (Section 4.2)
4. The *Arbitrator* arbitrates memory assigned to various pools by the Initializer in order to ensure reliability and low GC overheads. It also calculates a utility score for the resulting configuration. (Section 4.3)
5. Finally, the best settings for each of the probed container configurations are ranked by *Selector* based on their utility score and the best is returned as the final recommendation.

4.1 Statistics Generation

We use Thoth [21] framework to obtain a profile of the application which includes the JVM GC profile [38] as well the resource usage timeline generated using IBM's PAT tool [36] for each container in addition to the application event log profile generated by the application framework. Table 5 lists the statistics for an application. We leave a detailed explanation of the statistics to the technical report [22], instead provide an example for illustration.

Example. *Statistics for the* PageRank *application studied in Section 3 are listed in the third column of Table 5. It can be noticed that the application has a high* Cache Storage *requirement indicated by a high M_c and a low H. Further, a high M_u indicates a high task memory footprint which makes the application susceptible to out-of-memory errors.*

Importance of full GC events: In case the provided application profile contains no full GC events (significant of an application with very low memory footprint), estimating M_u accurately becomes hard. One solution is to base the calculations on maximum Old pool occupancy. This approach, though, leads to an over-estimation of task memory requirements and in effect, *sub-optimal*, albeit *reliable* recommendations provided by the RelM tuner. An empirical analysis is included in [22]. Based on the empirical evidence, we discard using Old pool occupancy to estimate M_u. Instead, we recommend simple changes to the configuration used for profiling based on three practical heuristics for increasing GC pressure: (a) Decrease Heap Size, (b) Increase Task Concurrency, and (c) Increase NewRatio. The new profile generated using the heuristics is expected to contain *full GC* events, making it more suited to the RelM tuner.

4.2 Initializer

We use the statistics presented in Table 5 to configure each memory pool for a given container configuration identified by the Containers per Node n and the Heap Size of each m_h. Notation of small letters is used to differentiate the test configuration from the profiled configuration. A safety factor δ denotes a fraction of memory to be kept unassigned. It acts as a safeguard against *out-of-memory* errors. The Initializer uses analytical models to configure each of Cache Storage, Task Shuffle, and Task Unmanaged independently. Memory pressures and potential GC bottlenecks in the resulting configurations are handled by the *Arbitrator* module later.

Cache storage. Cache Storage requirement is determined by scaling the maximum cache storage observed in the application profile by the cache hit ratio number.

$$m_c = m_h * \min\Big(\frac{M_c}{H * M_h}, 1 - \delta\Big) \tag{1}$$

Shuffle memory. We estimate Task Shuffle by scaling the maximum shuffle memory observed in the application profile by the data spillage fraction. It is assumed that each concurrently running task is an equal contributor to the spillage.

$$m_s = \min\Big(\frac{M_s}{1 - S/P}, (1 - \delta) * m_h\Big) \tag{2}$$

GC settings. The Old pool of JVM needs to be sized at least as big as the long term requirements, viz. M_i and m_c, in order to lower the GC overheads (Section 3.4). The GC parameter $NewRatio(NR)$ is set accordingly. Eden size is calculated by subtracting two survivor spaces specified by $SurvivorRatio(SR)$ from Young pool size.

$$\begin{gathered} NR = \text{ceil}\Big(\frac{M_i + m_c}{m_h - M_i - m_c}\Big) \\ m_o = m_h * \frac{NR}{NR + 1}, m_e = m_h * \frac{1}{NR + 1} * \frac{SR - 2}{SR} \end{gathered} \tag{3}$$

Task Concurrency. Task concurrency in a container is estimated based on the following stats obtained from the application profile: (a) avg CPU usage per task, (b) avg disk usage per task, and (c) max per-task memory required. The models assume a linear relation to obtain a conservative estimate.

$$\begin{gathered} p^{CPU} = \frac{1}{n}\frac{(1 - \delta) * 100}{CPU_{avg}/P}, p^{disk} = \frac{1}{n}\frac{(1 - \delta) * 100}{Disk_{avg}/P} \\ p^{mem} = \frac{(1 - \delta) * m_h}{M_u}, p = \min(p^{CPU}, p^{disk}, p^{mem}) \end{gathered} \tag{4}$$

Example. *The* PageRank *application studied in Section 3 when evaluated on the container configuration of* $n = 1$ *and* $m_h = 4404MB$, *with safety factor* $\delta = 0.1$, *generates:*

$$m_c = 3798MB, m_s = 0MB, p = 5, NR = 9 \tag{5}$$

4.3 Arbitrator

Building on the empirical analysis in Section 3, we build a general algorithm to tune a given configuration for reliability and low GC overhead. Algorithm 1 presents the pseudo-code.

Algorithm 1 RelM Arbitrator

Input: Configuration $\mathbf{c} = (M_i, M_u, p, m_c, m_s)$, Safety factor δ

```
1:  if (M_i + M_u) > (1 − δ) ∗ m_h then
2:      Return flagging insufficient memory
3:  end if
4:  while (M_i + p ∗ M_u + m_c) > m_o do
5:      one of the following three in a round-robin manner:
6:      I. Decrease p by 1 if p > 1
7:      II. Reduce m_c by M_u ensuring that m_c > 0.
8:         Change GC pools using Equation 3.
9:      III. Increase m_o by M_u ensuring m_o < (1 − δ) ∗ m_h
10: end while
11: Set shuffle memory m_s = min(m_s, 0.5 ∗ m_e/p)
12: Set output C = (M_i, M_u, p, m_c, m_s)
13: Set utility score U_C = (M_i + m_c + p∗(M_u + m_s)) / m_h
14: Return (C, U_C).
```

Line 1 checks if the configuration satisfies the bare minimum requirement of a container running at least one task at any given time. Lines 4-10 represent the main loop where actions to change configuration are carried out if the combined memory consumed by Code Overhead, Cache Storage, and Task Unmanaged exceeds Old. Please recall that the task memory values are obtained by profiling *full GC* events and correspond to the task objects tenured to Old. If the combined memory exceeds m_o, we perform one of the three actions given in Lines 6, 7, and 9 in a round-robin manner:

- Decrease Task Concurrency by 1. This reduces the memory footprint by M_u.
- Decrease Cache Capacity by M_u. We also adjust GC pools so that Old pool is just larger than the value $M_i + m_c$. The idea is to probe if an optimal GC setting for the given Cache Storage value can ensure safety as well.
- Increase old generation pool size by M_u. This optimization trades-off performance to ensure safety against out-of-memory errors (Recall *Observation 6* from Section 3.4).

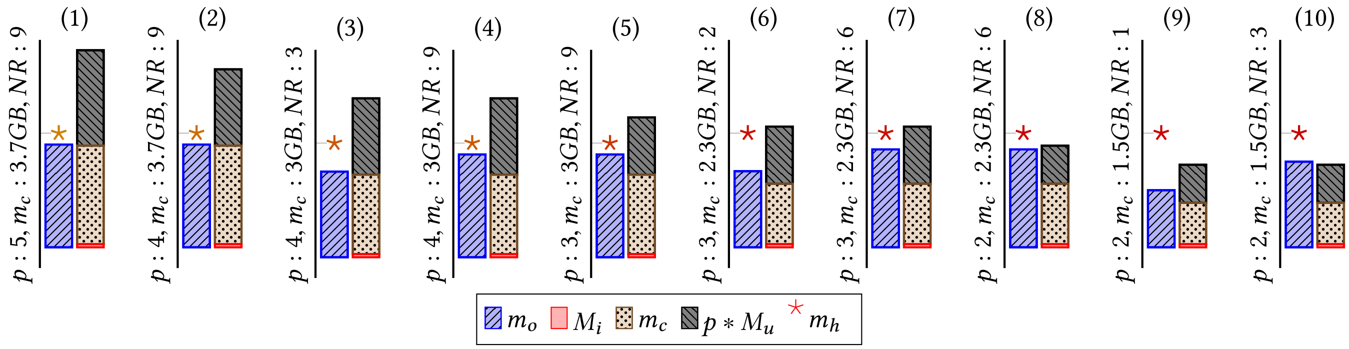

Figure 11: Working example showing steps of RelM's *Arbitrator* algorithm on PageRank application

At the end of the loop, settings for Task Concurrency, Cache Capacity, and NewRatio are locked in. Based on the available Eden, Task Shuffle is tuned in Line 11 which avoids the high GC overheads explained in Figure 8. Finally, Line 13 computes a utility score U which corresponds to the fraction of Heap allocated to the internal memory pools.

Example. *Continuing with the* PageRank *example for which the configuration produced by the initializer is given in Eq. 5. Figure 11 details the changes in memory pools starting with the initial configuration shown in (1). After 9 iterations of the main, a reliable configuration is found which sets Task Concurrency = 2, Cache Capacity = 1.5GB, and NR = 3. Compared to the profiled application run, this configuration lowers the cache capacity by 700MB. This, however, is not the only reliable configuration* RelM *finds: A better performing configuration is obtained when the process is repeated on a configuration of 2 Containers per Node. Section 6.2 presents this result.*

Analysis: As stated in RelM goals, *safety* is the primary objective. The Arbitrator meets this objective by ensuring that the combined allocation of internal memory pools remains within Heap. The next two performance objectives, a *high task concurrency* and a *high cache hit ratio*, are achieved by a two-phase process. *Initializer* first optimizes the Task Unmanaged and Cache Storage pools corresponding to the two requirements independently against the entire heap size. *Arbitrator* then takes small chunks out of the two pools in a round-robin manner until it can meet the *safety* condition. This process results in a *proportionally fair* [4] allocation for the two memory pools. The arbitration is invoked for each enumerated container configuration which is a small number because of the physical constraints in resource allocation. Within an invocation, the number of iterations of the main loop is a linear function of the maximum degree of parallelism (number of cores) in the worst case. So overall, the algorithm needs only a handful steps to recommend a configuration that best meets the goals.

5 BLACK-BOX TUNERS

AI-driven black-box formulation is a popular choice for autotuning because of its applicability to a wide variety of problem setups. The basic idea is to incrementally probe samples from the space of configurations to learn their impact on performance. We adopt two popular techniques to our problem: (1) A sequential model-based optimization called *Bayesian Optimization*, and (2) A model-free deep reinforcement learning algorithm called *Deep Distributed Policy Gradient*.

5.1 Bayesian Optimization

Bayesian Optimization [31] is a powerful learning technique which approximates complex response surface through adaptive sampling of the search space while balancing exploration (i.e., probing new regions) and exploitation (i.e., favoring the promising regions). At the core of BO is a surrogate model used to approximate the response surface. *Gaussian Process* [45] is an attractive choice because of its salient features such as confidence bound on predictions, support for noisy observations, and its use gradient-based methods [48].

We are given a data analytics application A and d parameters $x_1, x_2, \ldots, x_d$ to tune. The parameters are listed in Table 1. The performance metric, denoted by y, corresponds to the wall-clock duration of the application A on a setting $(x_1, x_2, \ldots, x_d) \in \mathcal{X}$. Tuning is carried out by adaptively collecting samples $\langle \mathbf{x}, y \rangle = \langle x_1 = v_1, x_2 = v_2, \ldots, x_d = v_d, y = p \rangle$. The prior belief in Gaussian Process is modeled as $f(\mathbf{x}) \sim GP(\mu_0, k)$, where $\mu_0 : \mathcal{X} \to \mathbb{R}$ denotes the prior mean function and $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ denotes the covariance function. Given n sampled points $\mathbf{x}_{1:n}$ and noisy observations $y_{1:n}$ (σ^2 denoting a constant observation noise), the unknown function values $\mathbf{f} := f_{1:n}$ are assumed to be jointly Gaussian, i.e. $\mathbf{f}|\mathbf{x} \sim \mathcal{N}(\mathbf{m}, \mathbf{K})$, and the observations $\mathbf{y} := y_{1:n}$ are normally distributed given $\mathbf{f}$, i.e. $\mathbf{y}|\mathbf{f}, \sigma^2 \sim \mathcal{N}(\mathbf{f}, \sigma^2\mathbf{I})$. The posterior mean and variance are then given by the following:

$$\begin{aligned} \mu_n(\mathbf{x}) &= \mu_0(\mathbf{x}) + \mathbf{k}(\mathbf{x})^\top(\mathbf{K} + \sigma^2\mathbf{I})^{-1}(\mathbf{y} - \mathbf{m}) \\ \sigma_n^2(\mathbf{x}) &= k(\mathbf{x}, \mathbf{x}) - \mathbf{k}(\mathbf{x})^\top(\mathbf{K} + \sigma^2\mathbf{I})^{-1}\mathbf{k}(\mathbf{x}) \end{aligned} \tag{6}$$

where $\mathbf{k}(\mathbf{x})$ is a vector of covariance between $\mathbf{x}$ and $\mathbf{x}_{1:n}$.

An acquisition function provided by BO suggests the next probe based on the posterior. Expected Improvement (EI), defined below, is a popular choice for the acquisition.

$$EI(\mathbf{x}; \mathbf{x}_{1:n}, y_{1:n}) = (\tau - \mu_n(\mathbf{x}))\Phi(Z) + \sigma_n(\mathbf{x})\phi(Z) \tag{7}$$

Here, τ denotes the current best, $Z = (\tau - \mu_n(\mathbf{x}))/\sigma_n(\mathbf{x})$, and Φ and ϕ are the standard normal cumulative distribution

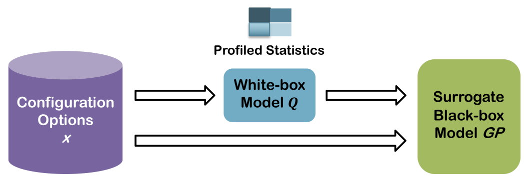

Figure 12: Design of Guided Bayesian Optimization

and density functions respectively. The next sample will try to balance *exploration*, captured by the uncertainty $\sigma_n(\mathbf{x})$, and *exploitation*, captured by $(\tau - \mu_n(\mathbf{x}))$. A combination of random sampling and standard gradient-based search is carried out to find the highest expected improvement [31].

5.2 Guided Bayesian Optimization

Recent work has shown that using execution profiles along with a knowledge of system internals can help speed up the tuning process significantly. Dalibard et. al. [10] propose Structured Bayesian Optimization (SBO) which lets system developers develop bespoke probabilistic models by including simple parametric models inferred from low-level performance metrics observed during a tuning run. Arrow [16], targeted at finding best VM configurations, augments a bayesian optimizer driven by VM characteristics with low-level performance metrics. Following in with the same philosophy, we design Guided Bayesian Optimization (GBO) to tune memory-based analytics applications.

Figure 12 shows the concept of GBO. The most important building block of GBO is a white-box model which is given a configuration and a set of profiled statistics for the application under test. The model outputs a set of derived metrics which is used in addition to the original configuration options for the optimization. The additional metrics are derived using simple analytical models with the purpose of separating out the most suitable region of configuration space from the more expensive region. Compared to SBO [10], which requires a system expert to design a parametric model by observing the system performance while tuning, GBO simplifies the process with a white-box model that can be used right from the beginning on any type of workload.

Guiding white-box model: The model used as a guide, Q, is based on the empirical analysis carried out in Section 3. Inputs to the model include: (a) Configuration options under test ($\mathbf{x}$), and (b) Profiled statistics from a prior execution, not necessarily using the same configuration (Table 5).

$$q_1^{\mathbf{x}} = \frac{M_i + \min(m_c^{\mathbf{x}}, m_c) + p^{\mathbf{x}} * (M_u + \min(m_s^{\mathbf{x}}, m_s)}{m_h^{\mathbf{x}}})$$
$$q_2^{\mathbf{x}} = \frac{M_i + m_c}{\min(m_o^{\mathbf{x}}, m_c^{\mathbf{x}})},\ q_3^{\mathbf{x}} = \frac{p^{\mathbf{x}} * \min(m_s^{\mathbf{x}}, m_s)}{0.5 * m_e^{\mathbf{x}}} \tag{8}$$
$$\mathbf{q}^{\mathbf{x}} = \{q_1^{\mathbf{x}}, q_2^{\mathbf{x}}, q_3^{\mathbf{x}}\}$$

Q generates three metrics as listed in Eq. 8. q_1 corresponds to the expected heap occupancy of a container. The numerator adds up the expected memory usage by every application level memory pool. The Cache Storage and Task Shuffle requirements (denoted by m_c and m_s) are modeled by Eq. 1 and Eq. 2 respectively. The intuition is to identify both the configurations under-utilizing memory (those with low scores) as well as the potentially *unsafe* ones (those with scores over 1). q_2 corresponds to the expected long term memory efficiency. Here, the numerator corresponds to the long term requirement while the denominator corresponds to the available long term memory storage considering the limits enforced by the configuration options. A high q_2 could mean either high disk overheads on account of data not fitting in memory or high GC overheads on account of data not fitting in Old pool. q_3 corresponds to the efficiency of the shuffle memory usage. Based on *Observation 7*, a high q_3 score implies a high GC overhead because of the large-sized data spills.

The set of metrics derived by model Q is designed to be the most practical means to identify safe, highly efficient, and low overhead configurations in accordance with the goals set out by RelM. This set could be expanded to add more indicators of the RelM goals in future.

Changes to surrogate model: The surrogate model is modified to fit metrics from model Q, i.e., $GP(\mathbf{x}_{1:n}, \mathbf{q}_{1:n}, y_{1:n})$. As before, the next probe is identified using the Expected Improvement score.

$$\mathbf{x}_{n+1} = \underset{\mathbf{x} \in \mathbf{X}}{\arg\max}\, EI(\mathbf{x}, \mathbf{q}^{\mathbf{x}}; \mathbf{x}_{1:n}, \mathbf{q}_{1:n}, y_{1:n}) \tag{9}$$

5.3 Reinforcement Learning

Reinforcement Learning (RL) involves an agent that interacts with an environment E in discrete timesteps. At each timestep t, it makes an observation, takes an action a_t, and receives a reward r_t. The action changes the state of the environment to s_t. We first map the terminology to our setup before describing the specific RL agent we use.

For the problem of tuning a given data analytics application, an *action* constitutes a change in configuration knobs (listed in Table 1). Similar to the approach used in CDBTune [60] for DBMS tuning, a *state* corresponds to a set of resource usage metrics. The statistics on CPU, IO, and memory usage listed in Table 5 constitute one half of the metrics. Following the philosophy of GBO, we add to this set the metrics derived from model Q (Eq. 8) to get a visibility into utilization of the internal memory pools. The *reward* function is borrowed from CDBTune as well; it considers the performance change at not only the previous timestep but also compared to the time tuning request was made.

DDPG Overview: Deep Deterministic Policy Gradient [28] is a policy-based model-free RL agent which combines Deep Q Neural Network with Actor-Critic models to work with continuous configuration parameter space. The DDPG actor learns a policy function $a_t = \mu(s_t|\theta^\mu)$, where θ^μ maps state s_t to value of action a_t. A critic function $Q(s_t, a_t|\theta^Q)$ evaluates the policy function estimated by the actor. Evaluation of

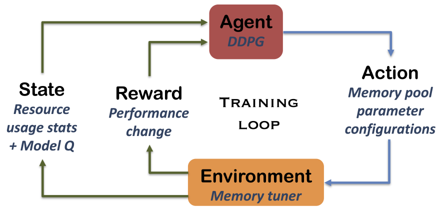

Figure 13: Reinforcement learning adopted for auto-tuning memory-based analytics

the value considers not only the current reward but also discounted future rewards. DDPG uses an *experience replay* memory to store the explored state-action pairs and uses a sample from the memory for learning its critic model.

DDPG, being a model-free algorithm, does not need to store all the combinations of states and actions it has explored. Exploration of action space is carried out by adding a noise sampled from a noise process $\mathcal{N}$ to the actor μ.

6 EVALUATION

6.1 Setup

Our evaluation uses two Spark clusters listed in Table 3. The applications we have picked represent Map and Reduce computations, machine learning, distributed graph processing, and SQL processing use cases. The test suite including input data sources is provided in Table 2. The input data is stored in HDFS co-located with the compute cluster. We have deliberately changed partition sizes for some of the applications from the default HDFS block size of 128MB to create another dimension of variability in the test suite.

Configuration Space. The maximum heap available for allocation per node is 4404MB on cluster A and 16GB on cluster B. We allow it to be distributed equally among 1, 2, 3, or 4 Containers per node. The number of concurrently running tasks on a node is limited by the number of physical cores. Therefore, the Task Concurrency value can range from 1 to the ratio of the physical cores to the number of containers. Cache Capacity and Shuffle Capacity values are set as a fraction (ranging from 0 to 1) of Heap. As Spark provides a unified memory pool [41] combining both Cache Storage and Task Shuffle, we set the capacity of the unified pool to the sum of Cache Capacity and Shuffle Capacity. The lowest possible value for NewRatio is 1. The maximum value, while unbounded in theory, is limited to 9 (giving 10% heap to the young pool) in our setup. SurvivorRatio is set to default.

Default Policy. The default configuration by Amazon EMR's *MaxResourceAllocation* policy is listed in Table 4.

Exhaustive Search. Our exhaustive search policy grids the configuration space by discretizing the domain of each parameter into 4 values. Only one of Cache Capacity and Shuffle Capacity is used depending on the dominant requirement of the application just to avoid collecting insignificant data. The minor memory pool capacity is set to 0.1. Despite the dimensionality reduction, *Exhaustive Search* is clearly an inefficient policy: The time taken to run all 192 configurations for an application on cluster A is at least 3 days.

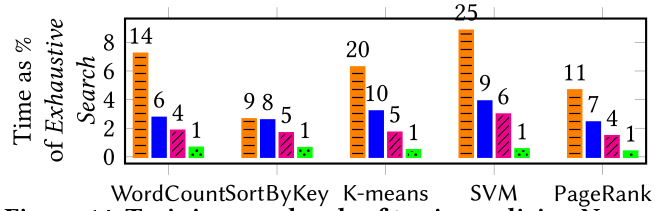

Figure 14: Training overheads of tuning policies. Number of iterations is shown on top of bars.

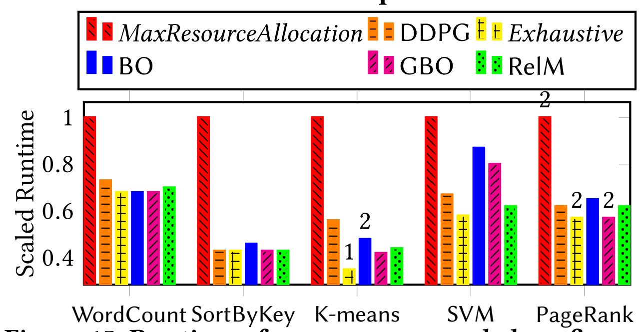

Figure 15: Runtime of every recommended configuration is scaled to the runtime of *MaxResourceAllocation*. Number of failed containers is shown on top of bars.

Black-box Policies. Bayesian Optimization (BO) is implemented using *scikit-learn* library in Python [43]. Since the accuracy of BO predictions depends on the number of samples explored, we bootstrap the model with 4 samples generated using Latin Hypercube Sampling [19] The objective function is set to the application runtime. If a run is aborted due to errors, the objective value for the sample is set to twice the worst runtime obtained on the samples explored so far. This heuristic ensures that the failing region is ranked low during exploration. The same setup is mimicked for our optimized policy of Guided Bayesian Optimization (GBO).

Reinforcement learning (DDPG) is another black-box policy we evaluate. DDPG algorithm described in Section 5.3 is implemented using PyTorch [42] library with its neural network parameters borrowed from CDBTune [60].

White-box Policy. RelM is our white-box model. Modules *Initializer* and *Arbitrator* are implemented in Java with ≈ 200 lines of code; the source is available online [39]. The safety fraction δ is set to 0.1 throughout.

6.2 Quality of Results

The first question we want to answer is ***how long does it take to produce high quality tuning results?***. We carry out *Exhaustive Search* on Cluster A and use it as a baseline for other policies. The black-box policies are trained on each application individually until they find a configuration with performance within top 5 percentile of the baseline. The process is repeated 5 to 10 times and only the mean values of overheads are plotted in Figure 14.

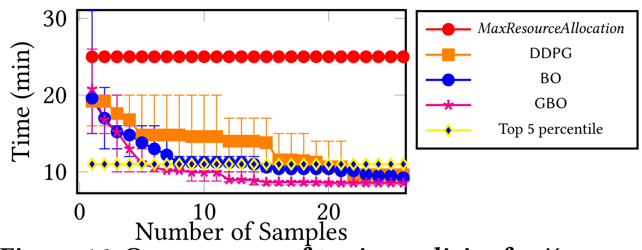

Figure 16: Convergence of tuning policies for K-means. Each tuner is run 5 times; the mean, min, and max values for the lowest runtime observed so far are plotted.

Table 6: Recommendations made for K-means

Policy	Containers per Node	Task Concurrency	Cache Capacity	New Ratio
Exhaustive	3	2	.8	7
DDPG	1	4	.6	4
BO	3	1	.75	3
GBO	3	1	.8	5
RelM	2	2	.68	4

Table 7: Analysis of a BO run for SVM.

Sample #	Containers per node	Task concurrency	Cache capacity	New Ratio	Runtime (minutes)
0	1	4	0.6	7	8.5
0	2	1	0.4	3	9.3
0	3	2	0.2	5	7.1
0	4	2	0.8	1	13
1	4	2	0.2	5	7.3
2	2	3	0.2	7	7.5
3	3	2	0.2	3	6.6
4	3	2	0.2	1	**6.5**
5	2	3	0.2	1	6.7
6	2	4	0.2	1	7

Table 8: Comparing tuning algorithm overheads

Component	DDPG	BO	GBO	RelM
Statistics Collection	5ms	1ms	5ms	5ms
Model Fitting	100ms	140ms	180ms	0.1ms
Model Probing	2ms	800ms	1500ms	0.02ms
Model Size	3Kb	5Kb	6Kb	-

RelM needs a single application run in each case to analytically find a desired configuration. So it has the lowest overhead. The regression policies, BO and GBO, require less than 4% of the effort needed for *Exhaustive Search* with GBO being about 2 times faster. The DDPG policy takes longer, but still less than 10% time compared to the exhaustive search.

Compared to BO, DDPG can be seen to take longer to optimize. Figure 16 shows an example run showing how the policies converge. For the first 12 iterations, DDPG tries out configurations with lower values for Cache Storage with very low rewards. Post which, it starts exploring higher values for cache, higher rewards follow, and the model converges to the desired performance. Between BO and GBO, we observe that GBO model fits data earlier compared to BO. Our technical report [22] carries out an analysis using a validation set which corroborates the observation.

The second question we want to analyze is ***how much performance improvement is exhibited by our tuning policies?***. We use a stopping criteria for black-box exploration policies: Bayesian policies are executed until the expected improvement falls below 10% and at least 6 new samples have been observed in addition to the 4 LHS samples [3]; DDPG is similarly stopped when it has observed 10 new samples. Although both the policies are capable of re-using models from prior tuning runs, we train them with a cold start in this evaluation; model re-use is discussed in Section 6.5.

Figure 15 compares the performance. RelM consistently achieves a runtime within 10% of the best configuration found using *Exhaustive Search*. Moreover, RelM ensures no containers run out of memory. Table 6 lists the recommendations for K-means. It can be noted that a high memory is allocated to Cache Storage by the policies of *Exhaustive*, BO, and GBO leaving very small memory for other objects thereby risking out-of-memory failures.

The performance improvement over the default setup, in most cases, is between 50%-70%. In the case of SVM, however, BO and GBO policies find configurations that are better than the default ones by only 10% and 20% respectively. This happens due to exploration hitting a local minima.

Black-box models can get stuck in a local minima. We saw an example earlier where DDPG took a long time to explore a region outside a local optima. Here, we note that the quality of results of BO, to a large extent, depends on the initial samples used in bootstrapping. We provide a log of BO run for SVM in Table 7. Based on the initial samples, BO pins down the Cache Capacity to 0.2 and continues exploring the other parameters. The application requires a capacity over 0.5 to fit in the entire cached data in memory. While this fact is captured in the white-box models of RelM, BO fails to explore this region. GBO, though not exempt, tends to come out of a local minima quicker because of the additional features from model Q guiding the exploration.

6.3 Algorithm Overheads

Overheads presented in Figure 14 are largely dominated by observation (stress testing) times. We focus on the other components here, viz. (1) Statistics collection, (2) Model fitting, and (3) Model probe. Table 8 compares one iteration from each algorithm. Except BO, all algorithms involve collecting internal resource usage statistics to build either white-box models or state metrics.

While model fitting involves an update of the actor-critic networks in DDPG, it requires an update of Gaussian Process with a new observation in BO. The higher overhead for GBO compared to BO is down to the added dimensionality due to model Q. The same is true when probing the model which involves computing expected improvement on a sample of configuration space. These numbers show why the BO regression model is not suited for high dimensional spaces.

In the case of RelM, both model fitting, which evaluates a small series of analytical models, and model probe, which

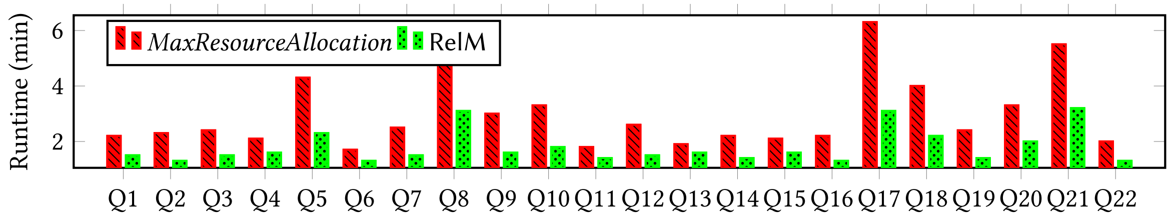

Figure 17: Performance comparison of TPC-H Queries run using *MaxResourceAllocation* policy and using RelM.

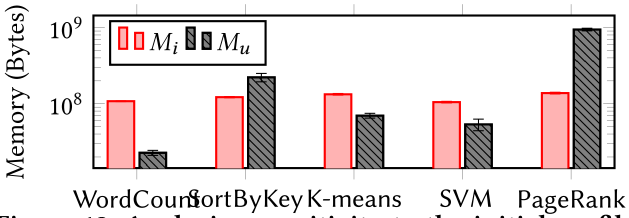

Figure 18: Analyzing sensitivity to the initial profile by invoking RelM with 16 unique initial profiles. Error bars indicate the standard error of the mean.

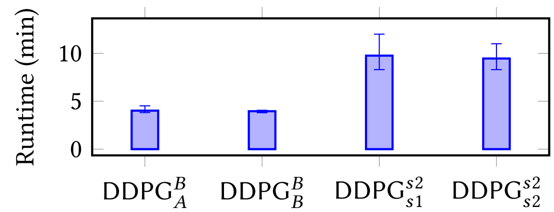

Figure 19: Studying generality of DDPG by applying it to a different cluster and to a different input dataset

involves looping through a small handful of container configurations, are inexpensive. We performed a small scalability test by artificially creating 100 container configurations, a considerably large number compared to the practical cluster setups. The model probe time goes up to 10ms which, though is a considerable increase from our test environment, is a small overhead when compared to other algorithms.

The black-box models can be saved for later use if an application similar to previously seen one is to be tuned. We compare the storage overhead of the models for the same. While DDPG stores the learned parameters of the actor-critic neural network, BO stores the entire training data. Though the last row of Table 8 shows that both DDPG and BO have a small storage overhead, the size of BO model grows linearly with training data making it suitable only when the number of samples used for training is small.

6.4 Analysis of RelM

As shown in Figure 15, RelM can handle different workload types. We also evaluate TPC-H benchmark workload on Cluster B to further press the point. As seen from Figure 17, the workload when executed using *MaxResourceAllocation* takes a total of 66 minutes. Using profile of this run, RelM cuts the runtime down to 40 minutes, a saving of 40%.

Section 4.1 discussed the importance of *full GC* events in the profile. We include a plot in Figure 18 showing the variability in the estimates of Code Overhead (M_i) and Task Unmanaged (M_u) pools by using different profiles containing full GC events. It is shown that the estimated memory requirements have little variance. The algorithm, as a result, recommends the same (with minor changes) configuration.

6.5 Generality of models

We analyze how our tuning policies adapt to a new environment or a new workload. As RelM takes a profile-based white-box approach to tuning, it needs at least a single run in the test environment. We have shown that a single profiling run is often sufficient as well since it contains enough information of the expected memory usage of both the resource containers and internal memory pools. Adaptability of RelM is evident from tests carried out on varied computational patterns, data layout (partition size), and resource clusters.

Black-box tuning policies, however, need to find ways to generalize models in order to reduce the stress testing time. OtterTune [2] re-uses Bayesian model trained on a prior workload by mapping the present workload based on the measurements of a set of external performance metrics. The OtterTune strategy is replicated in our setup by matching two applications based on the performance statistics (shown in Table 5) derived on the default configuration. However, the saved regression models cannot be adapted to the changes in hardware configuration and input data.

Unlike the performance-based regression model of BO, the DDPG model is trained using reward-feedback. It, therefore, showcases better adaptability to changes in test environment. We present an evaluation in Figure 19. First, we use a model trained on Cluster A to cross test the same workload, SVM application in this case, on Cluster B (denoted by $DDPG_A^B$). Its output is compared with the output produced by a model trained on Cluster B ($DDPG_B^B$). The cross testing is allowed to use only 5 test samples. By using the insights gained during prior training, the DDPG policy can quickly adapt to the hardware changes. Another experiment carried out by changing the input data scale factor for SVM workload on Cluster B (from s1 to s2) shows similar observation.

Summary

The tuning policies we evaluated each have their strong points. The evaluation justifies our approach of modeling interactions between the memory configuration options using which RelM model provides a good recommendation very quickly. Bayesian regression policies can provide optimality guarantees at a higher training cost; with GBO speeding up the exploration by 2x. DDPG policy supports an equally powerful AI-based tuning with minimal algorithm overheads and better adaptability to changes in environment making it an attractive choice for tuning problems where no simple white-box models can assist.

7 RELATED WORK

There has been a large body of work on auto-tuning the physical design of database systems [7] which includes index selection [8], data partitioning [44], and view materialization [1]. Comparatively less work has looked at on auto-tuning the internal configuration parameters like memory pools. Most commercial database systems provide configuration tuning wizards to DBAs which, based on the user feedback on workload performance, suggest better settings for configuration parameters using white-box models [24]. DB2 provides a Self-tuning Memory Manager (STMM) [50] which uses analytical models to determine cost-benefits of the internal memory pools. Oracle's *ADDM* can identify performance bottlenecks due to misconfigurations and recommend necessary changes [11].

Recent attempts at auto-tuning systems have either focussed on building *What-If* performance models [15, 49, 51] or, more popularly, on training ML-based performance models [2, 13, 29, 53–55, 60]. These models are trained either using a small-scale benchmark test bed, historical profiles, or from application performance under low workload. We argue that developing models that cater to changing workload or system environment is either impractical or potentially involves an expensive *online* learning cycle.

Black-box approaches are often employed to build an understanding of the interactions among configuration options on a newly seen workload. Many search-based approaches exist that use a combination of random sampling and local search [5, 14, 27, 57, 61]. However, such approaches are not suitable for our setup since there is a very high cost associated with running each experiment. Sequential Model-based Optimization (SMBO) approach [17] helps speed up the exploration by using a surrogate model to fit existing observations and using it to recommend the next probe. Bayesian optimization [31] is a powerful state-of-the-art SMBO technique that has found applications in many system tuning problems [2, 3, 6, 12, 16, 20]. We adapt the Bayesian Optimization using Gaussian Process [45] surrogate model for our problem setup. Alternate surrogate models such as Random Forest and Boosted Regression Trees have been shown to be better at modeling the non-linear interactions [16]. However, they lack theoretical guarantees on the confidence bounds that Gaussian Process offers. Also we did not find much qualitative difference among the models when evaluated in our setup and, therefore, do not include the results.

Guided Bayesian Optimization (GBO) we have developed is heavily motivated by Structured Bayesian Optimization (SBO) [10] which lets the system developers add structure to the optimization by means of bespoke probabilistic models consisting of a non-parametric bayesian model and a set of evolving parametric models inferred from low-level performance metrics. In comparison, GBO simplifies the process with a white-box model that can be used from the beginning of the tuning process on any workload. Another recent work targeted at finding the best VM configurations [16] augments a bayesian optimizer with low-level performance metrics though without building any analytical models.

Reinforcement learning is a powerful AI technique which is being adapted by database researchers for traditional problems such as query optimization [30] and database tuning [26, 60]. While both CDBTune and QTune use DDPG for database tuning, QTune adds a featurization step for SQL query workload to build models specific to the workload. We use DDPG in a similar manner, though without using featurization, since our goal is to tune each application individually.

We have focussed at the memory management options in data analytics workloads. Most cloud-based deployments provide robust settings that are expected to generalize well across applications. As an example, Amazon's Elastic MapReduce (EMR) provides a default policy for resource allocation on Spark clusters, called *MaximizeResourceAllocation* [34]. We establish through a thorough empirical analysis that the framework defaults do not generalize well and leave a lot of scope for performance improvements, a fact also shown by others [21, 47]. Like ours, there have been a few recent notable attempts at a systematic empirical analysis of data analytics systems. Charles Reiss [46] carried out an extensive evaluation of memory management in Spark and developed a tool to provision cluster memory to satisfy maximum memory requirements. Iorgulescu et. al. [18] studied memory elasticity in Hadoop, Flink, Spark, and Tez frameworks and used it to improve cluster scheduling. Both papers analyze each memory pool individually unlike RelM which also considers the interactions amongst the pools at multiple levels.

8 NEED FOR DATABASE PERFORMANCE DATA SCIENTISTS

We studied the problem of autotuning the memory allocation for data analytics applications using a state-of-the-art, AI-driven, black-box approach and our new empirically-driven, white-box solution called RelM. We showed how RelM provides better quality results (in terms of the desired objectives of low wall-clock time and performance reliability) with minimal overheads. RelM's superior performance highlights that tuning algorithms developed by *Database Performance Data Scientists* who combine an understanding of the underlying database platform with the ability to develop data-driven algorithms must not be overlooked while building autonomous/self-driving data processing systems.

REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 496–505. http://dl.acm.org/citation.cfm?id=645926.671701

[2] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1009–1024. https://doi.org/10.1145/3035918.3064029

[3] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 469–482. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard

[4] A. K. Austin. 1982. Sharing a Cake. *The Mathematical Gazette* 66, 437 (1982), 212–215. http://www.jstor.org/stable/3616548

[5] Liang Bao, Xin Liu, and Weizhao Chen. 2018. Learning-based Automatic Parameter Tuning for Big Data Analytics Frameworks. *CoRR* abs/1808.06008 (2018). arXiv:1808.06008 http://arxiv.org/abs/1808.06008

[6] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. 2018. Towards Better Understanding of Black-box Auto-tuning: A Comparative Analysis for Storage Systems. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*. USENIX Association, Berkeley, CA, USA, 893–907. http://dl.acm.org/citation.cfm?id=3277355.3277441

[7] Surajit Chaudhuri and Vivek Narasayya. 2007. Self-tuning Database Systems: A Decade of Progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*. VLDB Endowment, 3–14. http://dl.acm.org/citation.cfm?id=1325851.1325856

[8] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 146–155. http://dl.acm.org/citation.cfm?id=645923.673646

[9] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. 2014. Reservation-based Scheduling: If You'Re Late Don'T Blame Us!. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. ACM, New York, NY, USA, Article 2, 14 pages. https://doi.org/10.1145/2670979.2670981

[10] Valentin Dalibard, Michael Schaarschmidt, and Eiko Yoneki. 2017. BOAT: Building Auto-Tuners with Structured Bayesian Optimization. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 479–488. https://doi.org/10.1145/3038912.3052662

[11] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. 2005. Automatic Performance Diagnosis and Tuning in Oracle. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 84–94. http://cidrdb.org/cidr2005/papers/P07.pdf

[12] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. *PVLDB* 2, 1 (2009), 1246–1257. https://doi.org/10.14778/1687627.1687767

[13] Anastasios Gounaris and Jordi Torres. 2018. A Methodology for Spark Parameter Tuning. *Big Data Research* 11 (2018), 22 – 32. https://doi.org/10.1016/j.bdr.2017.05.001 Selected papers from the 2nd INNS Conference on Big Data: Big Data and Neural Networks.

[14] Herodotos Herodotou, Fei Dong, and Shivnath Babu. 2011. No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics *(SOCC '11)*. ACM, New York, NY, USA, Article 18, 14 pages. https://doi.org/10.1145/2038916.2038934

[15] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A Self-tuning System for Big Data Analytics. In *In CIDR*. 261–272.

[16] Chin-Jung Hsu, Vivek Nair, Vincent W. Freeh, and Tim Menzies. 2018. Arrow: Low-Level Augmented Bayesian Optimization for Finding the Best Cloud VM. In *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*. IEEE Computer Society, 660–670. https://doi.org/10.1109/ICDCS.2018.00070

[17] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*. Springer, 507–523.

[18] Calin Iorgulescu, Florin Dinu, Aunn Raza, Wajih Ul Hassan, and Willy Zwaenepoel. 2017. Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 97–109. https://www.usenix.org/conference/atc17/technical-sessions/presentation/iorgulescu

[19] Clarence Ireland. 1965. Fundamental Concepts In The Design of Experiments. *Technometrics* 7, 4 (1965), 652–653. https://doi.org/10.1080/00401706.1965.10490308 arXiv:https://www.tandfonline.com/doi/pdf/10.1080/00401706.1965.10490308

[20] Pooyan Jamshidi and Giuliano Casale. 2016. An Uncertainty-Aware Approach to Optimal Configuration of Stream Processing Systems. In *24th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2016, London, United Kingdom, September 19-21, 2016*. IEEE Computer Society, 39–48. https://doi.org/10.1109/MASCOTS.2016.17

[21] Mayuresh Kunjir and Shivnath Babu. 2017. Thoth in Action: Memory Management in Modern Data Analytics. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1917–1920. https://doi.org/10.14778/3137765.3137808

[22] Mayuresh Kunjir and Shivnath Babu. 2020. Black or White? How to Develop an AutoTuner for Memory-based Analytics [Extended Version]. arXiv:cs.DC/2002.11780

[23] Mayuresh Kunjir, Brandon Fain, Kamesh Munagala, and Shivnath Babu. 2017. ROBUS: Fair Cache Allocation for Data-parallel Workloads. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 219–234. https://doi.org/10.1145/3035918.3064018

[24] Eva Kwan, Sam Lightstone, K. Bernhard Schiefer, Adam J. Storm, and Leanne Wu. 2003. Automatic Database Configuration for DB2 Universal Database: Compressing Years of Performance Expertise into Seconds of Execution. In *BTW 2003, Datenbanksysteme für Business, Technologie und Web, Tagungsband der 10. BTW-Konferenz, 26.-28. Februar 2003, Leipzig (LNI)*, Gerhard Weikum, Harald Schöning, and Erhard Rahm (Eds.), Vol. 26. GI, 620–629. http://subs.emis.de/LNI/Proceedings/Proceedings26/article665.html

[25] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[26] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2118–2130. https://doi.org/10.14778/3352063.3352129

[27] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R Butt, and Nicholas Fuller. 2014. Mronline: Mapreduce online performance tuning. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 165–176.

[28] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2016. Continuous control with deep reinforcement learning. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1509.02971

[29] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16)*. ACM, New York, NY, USA, 50–56. https://doi.org/10.1145/3005745.3005750

[30] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (July 2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

[31] J. Mockus. 1989. *Bayesian approach to global optimization: theory and applications*. Kluwer Academic. https://books.google.com/books?id=FknvAAAAMAAJ

[32] Online. 2015. Juggling with Bits and Bytes. https://flink.apache.org/news/2015/05/11/Juggling-with-Bits-and-Bytes.html [Online; accessed 10-July-2019].

[33] Online. 2016. What Ails Spark in production? https://www.pepperdata.com/2016/07/11/ails-spark-production/ [Online; accessed 10-October-2018].

[34] Online. 2019. Amazon EMR Documentation. https://amzn.to/2zrpNtt [Online; accessed 10-July-2019].

[35] Online. 2019. Flink Configuration. https://ci.apache.org/projects/flink/flink-docs-stable/ops/config.html [Online; accessed 10-July-2019].

[36] Online. 2019. Intel's Performance Analysis Tool. https://github.com/intel-hadoop/PAT [Online; accessed 10-July-2019].

[37] Online. 2019. Java Garbage Collection Basics. https://bit.ly/2N8JyOp [Online; accessed 10-July-2019].

[38] Online. 2019. Java Management Extensions (JMX). https://bit.ly/2KIvbNn [Online; accessed 10-July-2019].

[39] Online. 2019. RelM Source code. https://github.com/dukedbgroup/global_log [Online; accessed 10-February-2020].

[40] Online. 2019. Spark Configuration. https://spark.apache.org/docs/latest/configuration.html [Online; accessed 10-July-2019].

[41] Andrew Or. 2016. Deep Dive: Apache Spark Memory Management. https://bit.ly/2C2x1YH [Online; accessed 10-July-2019].

[42] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.

[43] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[44] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. 2002. Automating Physical Database Design in a Parallel Database. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*. ACM, New York, NY, USA, 558–569. https://doi.org/10.1145/564691.564757

[45] Carl Edward Rasmussen. 2006. Gaussian processes for machine learning. MIT Press.

[46] Charles Reiss. 2016. *Understanding Memory Configurations for In-Memory Analytics*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-136.html

[47] Sandy Ryza. 2015. How-to: Tune Your Apache Spark Jobs (Part 2). https://bit.ly/2BuyZyS [Online; accessed 10-July-2019].

[48] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. 2016. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proc. IEEE* 104 (2016), 148–175.

[49] Juwei Shi, Jia Zou, Jiaheng Lu, Zhao Cao, Shiqiang Li, and Chen Wang. 2014. MRTuner: A Toolkit to Enable Holistic Optimization for Mapreduce Jobs. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1319–1330. https://doi.org/10.14778/2733004.2733005

[50] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. 2006. Adaptive Self-tuning Memory in DB2. In *Proceedings of the 32Nd International Conference on Very Large Data Bases (VLDB '06)*. VLDB Endowment, 1081–1092. http://dl.acm.org/citation.cfm?id=1182635.1164220

[51] Zilong Tan and Shivnath Babu. 2016. Tempo: Robust and Self-tuning Resource Management in Multi-tenant Parallel Databases. *Proc. VLDB Endow.* 9, 10 (June 2016), 720–731. https://doi.org/10.14778/2977797.2977799

[52] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*. ACM, New York, NY, USA, Article 5, 16 pages. https://doi.org/10.1145/2523616.2523633

[53] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 363–378. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/venkataraman

[54] Guolu Wang, Jungang Xu, and Ben He. 2016. A Novel Method for Tuning Configuration Parameters of Spark Based on Machine Learning. 586–593. https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0088

[55] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R Ganger. 2004. Storage device performance prediction with CART models. In *The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings.* IEEE, 588–595.

[56] Gerhard Weikum, Axel Moenkeberg, Christof Hasse, and Peter Zabback. 2002. Self-tuning Database Technology and Information Services: From Wishful Thinking to Viable Engineering. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*. VLDB Endowment, 20–31. http://dl.acm.org/citation.cfm?id=1287369.1287373

[57] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H. Xia, and Li Zhang. 2004. A Smart Hill-climbing Algorithm for Application Server Configuration. In *Proceedings of the 13th International Conference on World Wide Web (WWW '04)*. ACM, New York, NY, USA, 287–296. https://doi.org/10.1145/988672.988711

[58] Reynold Xin and Josh Rosen. 2015. Project Tungsten: Bringing Apache Spark Closer to Bare Metal. http://bitly.com/1KPpFBC [Online; accessed 10-July-2019].

[59] L. Xu, W. Dou, F. Zhu, C. Gao, J. Liu, H. Zhong, and J. Wei. 2015. Experience report: A characteristic study on out of memory errors in distributed data-parallel applications. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. 518–529. https://doi.org/10.1109/ISSRE.2015.7381844

[60] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. ACM, New York, NY, USA, 415–432. https://doi.org/10.1145/3299869.3300085

[61] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 338–350.